



Basic Research in Computer Science

BRICS RS-98-12 O. Danvy: Functional Unparsing

Functional Unparsing

Olivier Danvy

BRICS Report Series

RS-98-12

ISSN 0909-0878

May 1998

Copyright © 1998,

**BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK-8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide
Web and anonymous FTP through these URLs:**

**`http://www.brics.dk`
`ftp://ftp.brics.dk`**

This document in subdirectory RS/98/12/

Functional Unparsing ^{*}

Olivier Danvy

BRICS [†]

Department of Computer Science

University of Aarhus [‡]

November 1997 (revised in March and then in May 1998)

Abstract

A string-formatting function such as `printf` in C seemingly requires dependent types, because its control string determines the rest of its arguments.

Examples:

```
printf ("Hello world.\n");  
printf ("The %s is %d.\n", "answer", 42);
```

We show how changing the representation of the control string makes it possible to program `printf` in ML (which does not allow dependent types). The result is well typed and perceptibly more efficient than the corresponding library functions in Standard ML of New Jersey and in Caml.

^{*}Extended version of an article to appear in the Journal of Functional Programming.

A preliminary version appeared as BRICS RS-98-5.

[†]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

[‡]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.

Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55.

E-mail: danvy@brics.dk

Home page: <http://www.brics.dk/~danvy>

1 The Problem

In ML, expressing a printf-like function is not as trivial as in C. For example, we would like that evaluating the expression

```
format "%i is %s%n" 3 "x"
```

yields the string "3 is x\n", as specified by the pattern "%i is %s%n", which tells `format` to issue an integer, followed by the constant string " is ", itself followed by a string and ended by the newline character.

What is the type of `format`? In this example, it is

```
string -> int -> string -> string
```

but we would like our printf-like function to handle any kind of pattern. For example, we would like

```
format "%i/%i" 10 20
```

to yield "10/20". In that example, `format` is used with the type

```
string -> int -> int -> string
```

However, we cannot do that in ML: `format` can only have one type.

2 Analysis

The crux of the problem is that the type of `format` *depends* on the value of its first argument, i.e., the pattern. This has led, for example, Shields, Sheard, and Peyton Jones to propose a dynamic type system that makes it possible to express such a formatting function by delaying type inference until the pattern is available [3].

The culprit, however, is not ML's type system, but the fact that the pattern is represented as a string, which `format` in essence has to interpret (in the sense of a programming-language interpreter).

3 A Solution

Let us pursue this programming-language analogy, i.e., that `format` interprets the pattern. Instead of considering the *concrete syntax* of each pattern – as a string, we can consider its *abstract syntax* – as a data type.

Abstract syntax of patterns: The data type of patterns is composed of the following pattern directives:

- `lit` for declaring literal strings (`" is "` and `"/"` above);
- `eol` for declaring newlines (`%n` above);
- `int` for specifying integers (`%i` above); and
- `str` for specifying strings (`%s` above).

In addition, we provide the user with an associative infix operator `oo` to glue pattern components together.

Cosmetics: For cosmetic value, we could also provide two “outfix” directives `<<` and `>>` to delimit a pattern.

We could also define the operator `%` to be the polymorphic identity function, so that, e.g., `%int` (or even `%i` for that matter) would be a valid pattern directive.

Two examples: Thus equipped, we can make `format` construct an appropriate (statically typed) higher-order function, as in the following two examples.

```
format (int oo lit " is " oo str oo eol) : int -> string -> string
format (int oo lit "/" oo int) : int -> int -> string
```

The insights: Rather than making `format` interpret the pattern recursively, we make the pattern construct an appropriate higher-order function inductively. In that, we follow Harry Mairson’s observation that most of the time, our programs are inductive, not recursive [2]. More concretely, we use continuation-passing style (CPS) to thread the constructed string throughout. We also exploit the polymorphic domain of answers to instantiate it to the appropriately typed function. Formatting a string then boils down to supplying the initial continuation and the initial string.

For example, the type of the `eol` directive reads as follows.

```
(string -> 'a) -> string -> 'a
```

Its first argument is the continuation, which expects a string and yields the final answer. Its second argument is the threaded string, and because it is in CPS, this directive also yields the final answer.

For a second example, the type of the `int` directive reads as follows.

```
(string -> 'a) -> string -> int -> 'a
```

Its first argument is the continuation and its second argument is the threaded string. This directive yields a function expecting an integer and yielding the final answer.

The directives: `lit` and `eol` operate in a similar way:

```
fun lit x k s (* : string -> (string -> 'a) -> string -> 'a *)
  = k (s ^ x)
```

```
fun eol k s (* : (string -> 'a) -> string -> 'a *)
  = k (s ^ "\n")
```

As for `int` and `str`, they also operate in a similar way:

```
fun int k s (x:int) (* : (string -> 'a) -> string -> int -> 'a *)
  = k (s ^ (makestring x))
```

```
fun str k s x (* : (string -> 'a) -> string -> string -> 'a *)
  = k (s ^ x)
```

N.B. One can uncurry the directives and also change the order of their parameters, but the present formulation yields the simplest definition of `oo`.

Glueing the directives: We can implement `oo`, for example, as function composition (`o` in ML). So glueing `int` together with itself, for example, yields a function of the following type.

```
int oo int : (string -> 'a) -> string -> int -> int -> 'a
```

Initializing the computation: The job of `format` reduces to providing an initial continuation and an initial string to trigger the computation specified by the pattern:

```
fun format p (* : ((string -> string) -> string -> 'a) -> 'a *)
  = p (fn (s:string) => s) ""
```

So given the pattern `int oo int`, the `format` function supplies it with an initial continuation (the identity function over strings) and an initial string (the empty string), yielding a value of the following type, as desired.

```
int -> int -> string
```

4 An Alternative Solution

Alternatively, and given an end-of-pattern directive (implemented as the identity function), we can implement glueing as function application instead of as function composition. In both cases, the implementation of the directives remains the same, but the definition of `format` need no longer supply an initial continuation, since the initial continuation in effect is already provided by the end-of-pattern directive:

```
fun format' p (* : (string -> 'a) -> 'a *)
  = p ""

fun eod (s:string) (* : string -> string *)
  = s
```

Therefore, glueing `int` together with itself and the end-of-pattern directive, for example, yields a function of the following type.

```
int oo int oo eod : string -> int -> int -> string
```

More on cosmetics: Implementing glueing as function application makes it simple to implement the outfix directives `<<` and `>>` mentioned in Section 3. We can simply define each of them as the polymorphic identity function:

```
fun << x
  = x
fun >> x
  = x
```

And then we can write, e.g., the following:

```
<< int oo int >> : string -> int -> int -> string
format' (<< int oo int >>) : int -> int -> string
```

5 Assessment

Formatting strings is a standard example in partial evaluation [1]: the formatting function can be specialized with respect to any given pattern. Partial evaluation then removes the overhead of interpreting each pattern. So, for example, specializing a term such as

```
format (int oo lit " is " oo str oo eol)
```

yields the following more efficient residual term.

```
fn (x1:int) => fn x2 => (makestring x1) ^ " is " ^ x2 ^ "\n"
```

The required partial-evaluation steps can be very mild: for the functional specification described here, mere inlining (β -reduction) suffices. The back end of the ML Kit, for example, provides the specialization just above (Martin Elsmann, personal communication, March 1998).

Independently of partial evaluation, the functional specification is also efficient on its own. For example, besides being type-safer, it appears to be perceptibly faster than the resident `format` function in the New Jersey library `Format`: it is 3 to 4 times faster if glueing is implemented as function composition. Ditto for the resident `sprintf` function in the Caml library: the functional specification is 2 to 3 times faster if glueing is implemented as function composition. In both cases, making function composition left- or right-associative has little influence on the overall efficiency. Finally, implementing glueing as (right-associative) function application gives another 10% speedup both in Standard ML of New Jersey and in Caml.

Independently of efficiency, this functional specification of `format` further illustrates the expressive power of ML, or for that matter of any functional language based on the Hindley-Milner static type system [4]. It also easily scales up to inductive types such as lists.

```
fun lis t k s [] (* : ((string -> 'a) -> string -> 'b -> 'a) -> *)
  = k (s ^ "[]") (* (string -> 'a) -> string -> 'b list -> 'a *)
| lis t k s xs
  = let fun loop [x] k s
        = t (fn s => k (s ^ "[")) s x
        | loop (x :: xs) k s
        = t (fn s => loop xs k (s ^ ", ")) s x
      in loop xs k (s ^ "[")
    end
```

This new directive is parameterized with a type and is used as follows.

```
format (lis int oo lit " " oo lis (lis str))
: int list -> string list list -> string
```


Acknowledgements

Thanks to Richard Bird for editorial advice.

References

- [1] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
- [2] Harry Mairson. Outline of a proof theory of parametricity. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 313–327, Cambridge, Massachusetts, August 1991. Springer-Verlag.
- [3] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 289–302, San Diego, California, January 1998. ACM Press.
- [4] Zhe Yang. Encoding types in ML-like languages (preliminary version). Technical Report BRICS RS-98-9, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 1998.

Recent BRICS Report Series Publications

- RS-98-12 Olivier Danvy. *Functional Unparsing*. May 1998. 7 pp. This report supersedes the earlier report BRICS RS-98-5. Extended version of an article to appear in *Journal of Functional Programming*.
- RS-98-11 Gudmund Skovbjerg Frandsen, Johan P. Hansen, and Peter Bro Miltersen. *Lower Bounds for Dynamic Algebraic Problems*. May 1998. 30 pp.
- RS-98-10 Jakob Pagter and Theis Rauhe. *Optimal Time-Space Trade-Offs for Sorting*. May 1998. 12 pp.
- RS-98-9 Zhe Yang. *Encoding Types in ML-like Languages (Preliminary Version)*. April 1998. 32 pp.
- RS-98-8 P. S. Thiagarajan and Jesper G. Henriksen. *Distributed Versions of Linear Time Temporal Logic: A Trace Perspective*. April 1998. 49 pp. To appear in *3rd Advanced Course on Petri Nets, ACPN '96 Proceedings*, LNCS, 1998.
- RS-98-7 Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. *Marked Ancestor Problems (Preliminary Version)*. April 1998. 36 pp.
- RS-98-6 Kim Sunesen. *Further Results on Partial Order Equivalences on Infinite Systems*. March 1998. 48 pp.
- RS-98-5 Olivier Danvy. *Formatting Strings in ML*. March 1998. 3 pp. This report is superseded by the later report BRICS RS-98-12.
- RS-98-4 Mogens Nielsen and Thomas S. Hune. *Deciding Timed Bisimulation through Open Maps*. February 1998.
- RS-98-3 Christian N. S. Pedersen, Rune B. Lyngsø, and Jotun Hein. *Comparison of Coding DNA*. January 1998. 20 pp. To appear in *Combinatorial Pattern Matching: 9th Annual Symposium*, CPM '98 Proceedings, LNCS, 1998.
- RS-98-2 Olivier Danvy. *An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*. January 1998.
- RS-98-1 Olivier Danvy. *A Simple Solution to Type Specialization (Extended Abstract)*. January 1998. 7 pp.